CS764 Project Proposal (Project Idea #6)

Derek Paulsen 3/8/21

From the project ideas document

> Databases for tracking customer sales tend to suffer from lock contention as they scale up the number of concurrent new order creation. When a new order is created, a sequential order number is assigned to it. When the record is inserted, the index on the order number must be modified with the new order number. Because the new order number is the largest such number, it always goes to the rightmost leaf page of the B-tree index. Thus, that page is almost always locked for writes, and when splits occur it cascades into a tree rebalance at a high cost to concurrency. Design strategies for dealing with this problem automatically and implement them to compare their concurrency/throughput behavior.

This document will cover related work on this problem, and present a proposed a solution to it.

Related Work

The authors of PALM address having many threads updating and reading the B+-Tree index concurrently by using a bulk synchronous parallel (BSP) paradigm and batching reads and writes and doing bulk reads or bulk writes on the index. This approach allows the algorithm to avoid the use of latches which reduces the synchronization cost and increases throughput on the queries. While PALM is not intended to deal with insertion of sequential keys, it is likely to have benefits for such a workload since the one of the issues with inserting sequential keys is the overhead associated with latching (or locking) the furthest right leaf node.

Similar to PALM, B^{ε}-Trees use batching to improve throughput on write intensive workloads, however instead of immediately batching writes, each non-leaf node in the tree has some fixed buffer space allocated to it to allow store updates to the tree. When a buffer on a node becomes full, the updates are flushed to the child nodes. This approach allows for many update operations to be done at once to improve throughput at the cost of increasing the search time, since the buffers of each node of the path must be examined along with the keys in the nodes. This approach to batching could possibly improve performance of inserting sequential keys since the inserts can be batched, however the right most path on the tree will still experience high numbers of updates and hence is likely to still be a bottle neck. Additionally, the batch updates will only be a fraction of the B-tree's page size, meaning that the batches will potentially be much smaller than the optimal batch size.

Proposed Approach

Taking inspiration from B^{ε}-Trees and PALM, our solution batches writes to achieve better throughput. The high level solution has two major components, a B-tree (in this case the B^{link}-Tree), **T** and a in memory searchable thread-safe buffer, **S**. The

algorithm is then laid out below.

Insertion

Say we want to insert a key, k, into **T**. In order to do this, we first check and see what the highest key currently stored in **T** is, which we will call k'. If $k <= k'$, we simply insert k into **T** as normal. If $k > k'$, we first consult the buffer **S**, if **S** is full, we flush the inserted keys from **S** into **T** and update k' to the highest key that was in **S** and return, otherwise we insert k into **S**.

Search

To search for a key k we first check the high key in **T** (k'). If $k <= k'$, we do a normal search of **T** and return the results. If $k > k'$ we search **S** and return the result if it is found.

Implementation of T

B-Trees are in theory fairly simple data structures with well defined operations. In practice however they are complex to implement, especially when high concurrency is desired. Implementing a high performance B-Tree for experimentation would be beyond the scope of a course project so to get around this issue we will use an existing implementation. We propose using PostgreSQL's B-Tree implementation because it is known to be reasonably performant, implemented with high concurrency in mind, and is commonly used. To keep things simple we will leverage PostgreSQL B-Tree index via *standard SQL queries*, this is due to the PostgreSQL implementation assuming that there is a buffer manager and other utilities to manage page access, which makes pulling it out of the source code impractical. Furthermore, we were unable to find an implementation of a standalone B-Tree than allowed for the same level of concurrency provided by state of the art RDBMS's.

Implementation of S

The rationale behind this approach is to have a write optimized data structure that has reasonable read performance on a relatively small set of keys. This allows individual writes to be fast and updates to the B-tree be optimized via bulk updates while paying minimal cost on read performance. The natural question is then, how to implement **S** to achieve these properties. First, since there are multiple threads updating **S** we need a way of avoid concurrency issues, which there are two basic approaches, lock based data structures and lock-free data structures. Below we give a brief overview of two implementations of **S**, one lock based and one lock-free.

Baseline Implementation of S (Lock Based)

For this basic implementation we proposed using a single queue with a lock that can be acquired in shared or exclusive mode. For reading the lock would be acquired in shared mode and for writing the lock would be acquired in exclusive mode.

Lock-Free Implementation of S

The goal of this implementation of **S** is to adapt the lock based baseline version of **S** into a lock free algorithm. As with the baseline implementation, all threads will be writing to the same single data structure. There are many data structures that

can be implemented as a lock free algorithm but the simplest is a linked-list. Hence the baseline lock-free implementation of **S** will be a linked-list based queue that all threads write to.

Evaluation

To evaluate our proposed solution we will measure the performance of three different solutions on varying workloads. The solutions we will be comparing are,

1. Baseline, B^{link}-Tree only
2. Lock-based **S** with B^{link}-Tree
3. Lock-free **S** with B^{link}-Tree

For each of these solutions we will test various workloads,

1. Sequential key writes only
2. Sequential key writes with random reads
3. Sequential key writes with frequent reads of newer keys (keys that would likely be in **S**)

For each of these workloads we will measure the both throughput and query latency.